

Program Analysis

Foundations and Applications

Kristopher Micinski

kkmicins@syr.edu

(First task: send me an email)

Program analysis answers **these** kinds of questions...

“Can this program ever *potentially* crash due to a segmentation fault?”

“Will the value of this variable ever be 0?”

“What possible runtime type could this (Python) value be?”

“If I invoke this method, which concrete method will be invoked?”

“Does this program take the same amount of time to execute no matter what its input?”

“Will potentially-sensitive data ever be sent over the network?”

Pre requisites

- Know how to program, preferably functional programming
- *Ideally* some knowledge of programming languages
 - (E.g., lambda calculus, etc...)
- Willing to learn Racket, think about how code works
- Comfortable reading academic papers (or learning how)

Goals of This Course

- **Be able to read, understand, criticize, and conduct cutting-edge research in static analysis**
- Focus on program analysis principles (abstract interpretation, semantics, etc...)
- Allow exploration of applications
- Very few students = focus on everyone learning (almost) everything

Structure of Course...

- Rough slides. Largely us talking / working at board
- Will involve lots of programming
- Install Racket (we will use this to write analysis tools):
 - download.racket-lang.org
 - Install the command-line tools:
 - <https://docs.racket-lang.org/guide/cmdline-tools.html>

Course Structure: Part 1

- Week 1: Racket Intro, Metacircular Interpreters
 - HW 1: Intro Racket, metacircular interp
- Week 2: Lambda Calculus, Big-step interp for LC
- Week 3: Stack/Env passing, Church Encoding
 - HW 2: Stack-passing interp, church encoder
- Week 4: Abstract Machines, Continuations
- Week 5: CPS, CESK Machine
 - HW 3: CPS Transformation, abstract machines
- Week 6: Abstract Interpretation / Galois Connections
- Week 7: Abstracting Abstract Machines
 - HW 4: Basic AAM for Core Scheme

Take-Home Midterm (mix programming / written questions)

Course Structure: Part 2

- Week 8: Analysis sensitivity, polyvariance
- Week 9: Pushdown precision, call-return matching
 - HW 5: Implementing P4F, comparing analysis sensitivity
- Week 10: Object / field sensitivity
- Week 11: Constraint-based analysis
 - HW6: Object sensitivity, DOOP implementation
- Week 12: Symbolic execution, abstract symbolic execution
- Week 13+: Project Presentations

Take-home Final

Grading Breakdown

- 50% Projects / Homework (largely programming, some written)
- 40% (20% each) — Take-home midterm and Final
- 10% Class Participation. Course readings: 1/week, graded via a post to a mailing list

Grading Expectations

- B: actively participate in class, attempt all homework in good faith, >60% on all HW, >70% on midterm / final
- A: demonstrate deep understanding of at least a few topics in the course. >80% on homework, strong mastery on midterm / final
- <B: not coming to class, doing homeworks, etc...
- **Please feel free to email about your progress**

Grading Policies

- I may use an autograder (not sure yet)
- HW turned 0-48h late will get a 15% penalty
- HBW 48+h late will receive 30% penalty (but can be turned in until end of course)
- Reasonable exceptions made for things like paper deadlines, etc...

Office Hours

- Tentatively on Thursday mornings: 10:30-11:30
- Also by appointment
- Does this work for everyone?

This week...

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software Issues*, an American Mathematical Association Prospectus, July 1965.

A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework. This framework dictates the rules about the uses of user-coined names, and the conventions about characterizing functional relationships. Within this framework the design of a specific language splits into two independent parts. One is the choice of written appearances of programs (or more generally, their physical representation). The other is the choice of the abstract entities (such as numbers, character-strings, lists of them, functional relations among them) that can be referred to in the language.

The system is biased towards "expressions" rather than "statements." It includes a nonprocedural (purely functional) subsystem that aims to expand the class of users' needs that can be met by a single print-instruction, without sacrificing the important properties that make conventional right-hand-side expressions easy to construct and understand.

1. Introduction

Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things. The ISWIM (If you See What I Mean) system is a byproduct of an attempt to disentangle these two aspects in some current languages.

differences in the set of things provided by the library or operating system. Perhaps had ALGOL 60 been launched as a family instead of proclaimed as a language, it would have fielded some of the less relevant criticisms of its deficiencies.

At first sight the facilities provided in ISWIM will appear comparatively meager. This appearance will be especially misleading to someone who has not appreciated how much of current manuals are devoted to the explanation of common (i.e., problem-orientation independent) logical structure rather than problem-oriented specialties. For example, in almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. (Note that restrictions on what names can be coined also vary, but these are trivial differences. When they have any logical significance it is likely to be pernicious, by leading to puns such as ALGOL's integer labels.)

So rules about user-coined names is an area in which

Racket and FP

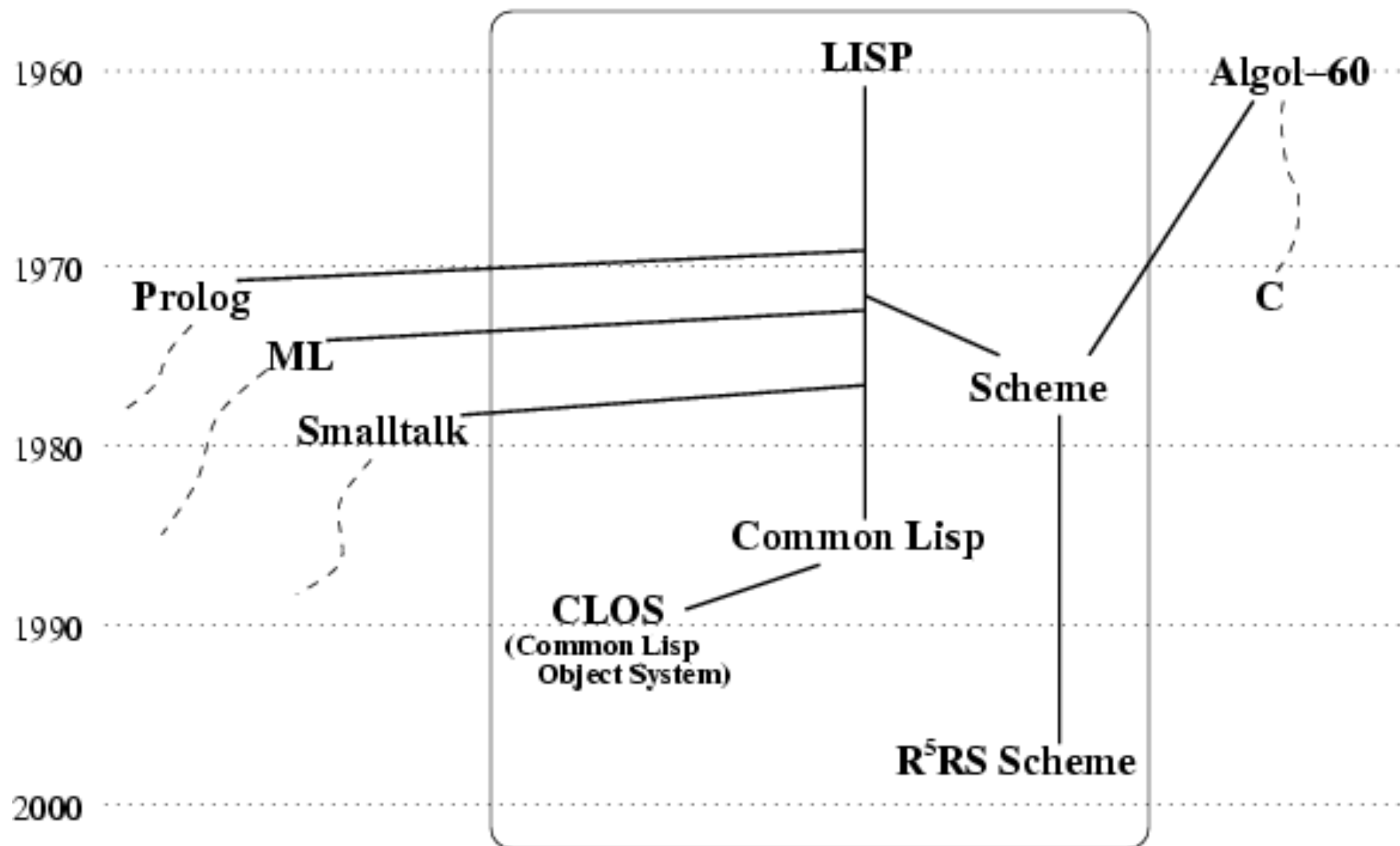
Meta language vs. Source Language

λ

Racket

- Dynamically typed: variables are untyped, values typed
- Functional: Racket emphasizes functional style
 - Immutability—Requires automatic memory management
- Imperative: Racket allows values to be strongly-updated, and is thus “impure” as functional languages go
 - Often discouraged
- Language-Oriented: Racket is really a language **toolkit**

A brief tour of history...



We wanted a language that allowed symbolic manipulation

List of either atoms or S-expressions

(this (is an) (s) expression)

List of either atoms or S-expressions

(this (is an) (s) expression)

List of either atoms or S-expressions

(**this** (is an) (s) expression)

↑
atom

List of either atoms or S-expressions

(this (is an) (s) expression)

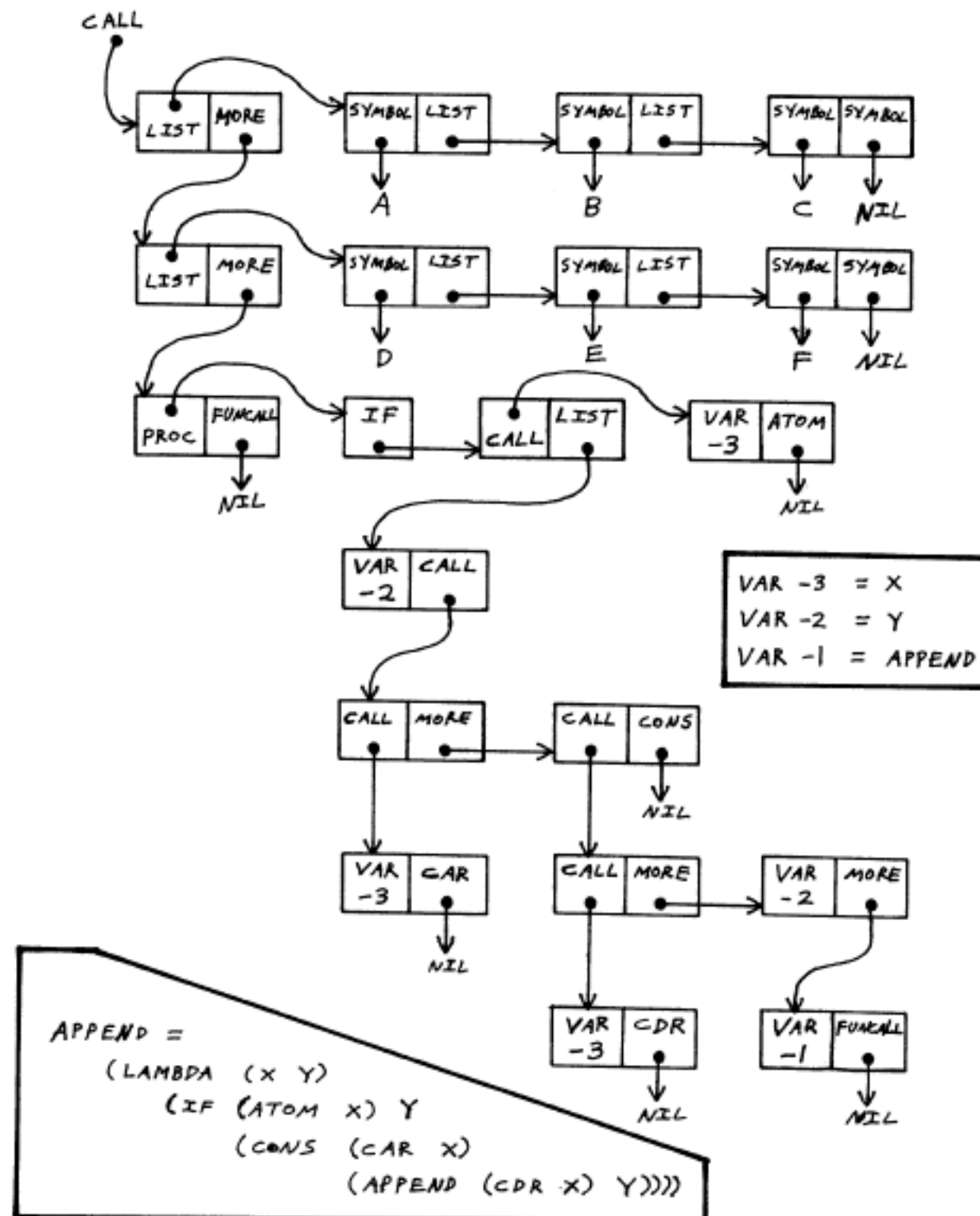
↑
S-expression

List of either atoms or S-expressions

(this (is an) (s) expression)

↑
also an S-expression

SIMPLE EXPRESSION FOR (APPEND '(A B C) '(D E F))





The First No-Compromise LISP Machine



 **LAMBDA**



So how do we write programs in this?

Calls `function` with arguments `arg0`, `arg1`, etc...

`(function arg0 arg1 ...)`

No infix operators! Everything is like this..

Two examples with + and -

Quiz problems: + and -

Calculate $(1 + (2 - 3)) - 4$

Introduce `if`, `and`, `or`

(if #t 1 2)

(if (equal? 2 3) 1 2)

(if (< 3 4) 1 2)

(if (and (or #t) #t) 1 2)

Notice: there is no “return” value

In functional programming, **every single expression**
implicitly returns its resulting value

(and #t #f)

(or #t ...)

Always true, even if ... doesn't terminate!

```
(define (factorial x)
  (if (equal? x 0)
      1
      (* (factorial (- x 1)) x)))
```

```
(define (factorial x)
  (if (equal? x 0)
      1
      (* (factorial (- x 1)) x)))
```

- Everything in parenthesis
- Prefix notation
- No variable assignment
- Recursion instead of loops
- No types
- No return

Quiz

- Compute the factorial of 5
- Compute the factorial of 20
- Compute the factorial of 20000

Quiz

- Define the fibonacci function:
 - Use `if`, `equal?`, `-`
- $\text{fib}(0) = 1$
- $\text{fib}(1) = 1$
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$

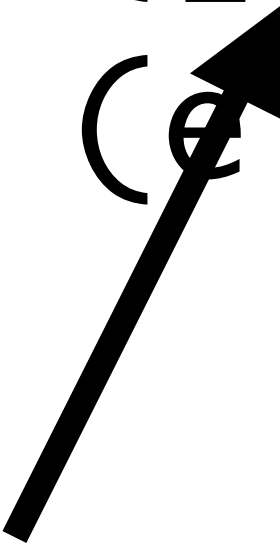
Introduce cond

Introduce cond

```
(cond  
  ([= x 1] 1)  
  ([= x 2] 2)  
  (else 3))
```

Introduce cond

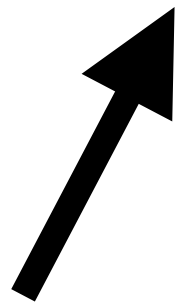
```
(cond  
  (= x 1) 1  
  (= x 2) 2  
  (else 3))
```



Any number of conditional “clauses”

Introduce cond

```
(cond  
  (= x 1) 1  
  (= x 2) 2  
  (else 3))
```



Potentially an “else” clause

Introduce cond


```
(cond  
  ([= x 1] 1)  
  ([= x 2] 2)  
  (else 3))
```

cond checks each clause and executes the body
of the first one that matches

If you get stuck, use the debugger...!

Racket is dynamically typed

```
> (length 2)
```



*length: contract violation
expected: list?
given: 2*

```
> |
```

```
(define (fib-again x)
  (cond
    [(< x 0) (raise 'lessthanzero)]
    [(eq? 0 x) 1]
    [(eq? 1 x) 1]
    [else 0]))
```

Define max

- cond
- <
- >
- equal?

Most Racket data is based on **lists**

`'(1 2 3)`

Most Racket data is based on **lists**

`'(1 2 3)`

`(first '(1 2 3)) → 1`

`(rest '(1 2 3)) → '(2 3)`

`(rest '(3)) → '()`

Can use **empty?** to check

`(empty? '())`

`(empty? '(1 2))`

Pronounced “empty-huh?”

Define max-of-list

- empty?
- first
- rest
- length?

Can create local variables with **let**

```
(let ([x 2]
      [y 3])
  (+ x y))
```

“Let x be 2 and y be 3 inside the expression...”

Quiz

Define (distance x1 y1 x2 y2)

Use `sqrt`

Use **let** at least once

You can create anonymous functions with lambda

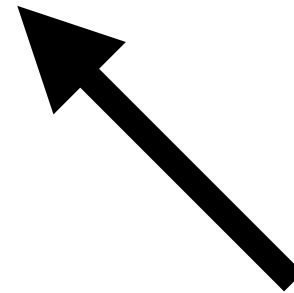
```
(lambda (x) (- x))
```

```
(lambda (str) (string-ref str 0))
```

```
((lambda (x) (* x x) 3)
```


```
(define f (lambda (x) (* 2 x))) (f 3)
```

```
(let ([x 1])  
  (+ x 1))
```



Rewrite this in terms of lambda!

Transform..

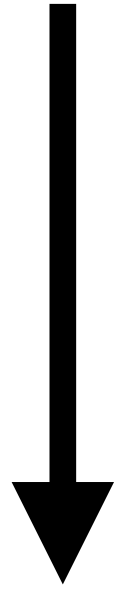
$(\text{let } ([x \ 1])$
 $(+ \ x \ 1))$  $((\text{lambda } (x)$
 $(+ \ x \ 1))) \ 1)$

Let is λ

```
(let* ([x 1]
        [y (+ x 1)])
  (list y x))
```

Lots of other things are λ too...

```
(define (f x) x)
```



shorthand for...

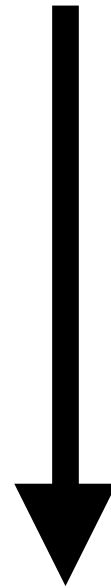
```
(define f (lambda (x) x))
```

`(define (f x) x)`



`(define f (lambda (x) x))`

`(define (f x y) x)`



`(define f (lambda (x y) x))`

...

Here's what most confused me...

```
> (lambda x x)
#<procedure>
> (lambda (x) x)
#<procedure>
> (lambda (x) x) 1
#<procedure>
1
> ((lambda (x) x) 1)
1
> ((lambda x x) 1)
'(1)
> |
```

Define hyphenate

```
> (hyphenate '("Kristopher" "Kyle" "Micinski"))  
"Kristopher-Kyle-Micinski"  
> |
```

(Use string-append)

Using higher order functions...

If you give me a function, I can use it

```
(define twice  
  (lambda (f)  
    (lambda (x)  
      (f (f x))))))
```

Challenge: figure out how I would use twice to add 2 to 2

Use Racket's add1 function

```
(add1 (add1 2))
```


All the forms we covered today:
Define, let, lambda, cond, if

Data Structures via Lists

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.



I WONDER IF THE CYCLES WILL CONTINUE FOREVER.



A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER'S PARENTHESSES



ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.



In today's class, we're going to build all data
from three things...

The first is **atoms**

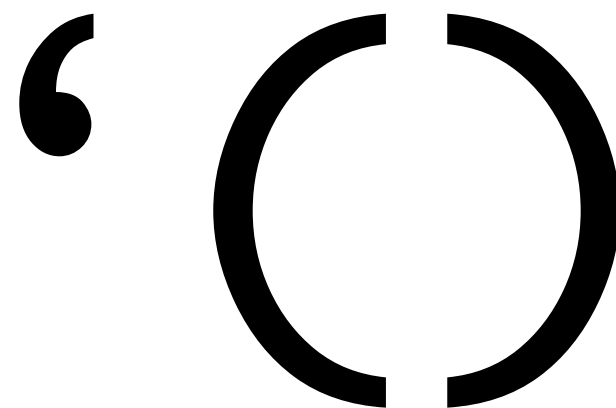
These are the *primitive things* in the language

‘symbol

1

These are like “int” and “char” in C++

The second is the **empty** list

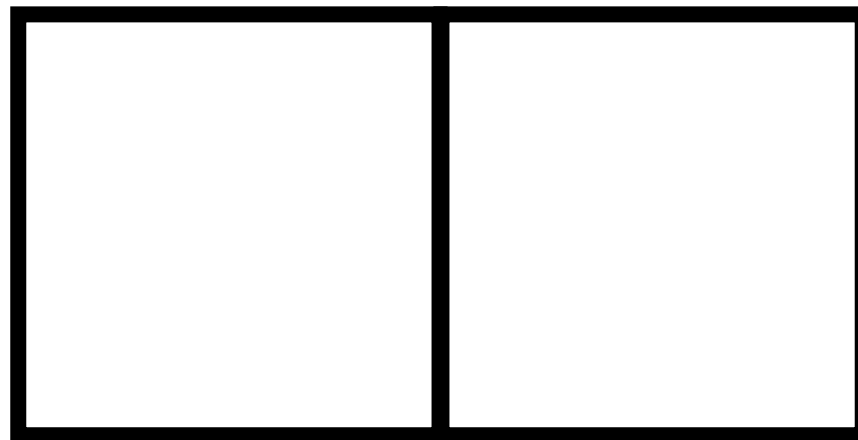


The last is **cons**

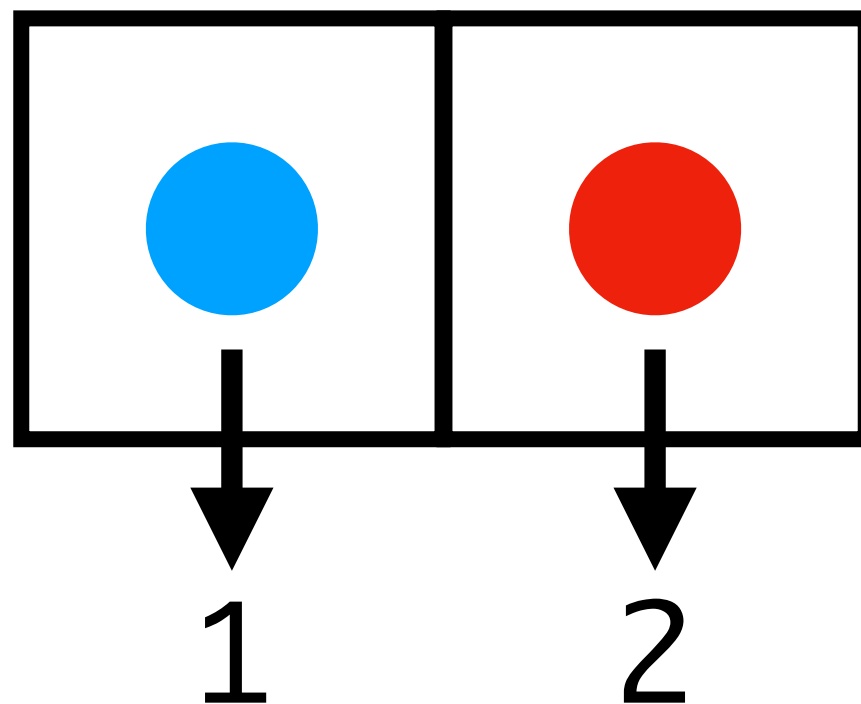
Cons is a function that takes two values and
makes a pair



That pair is represented as a **cons cell**

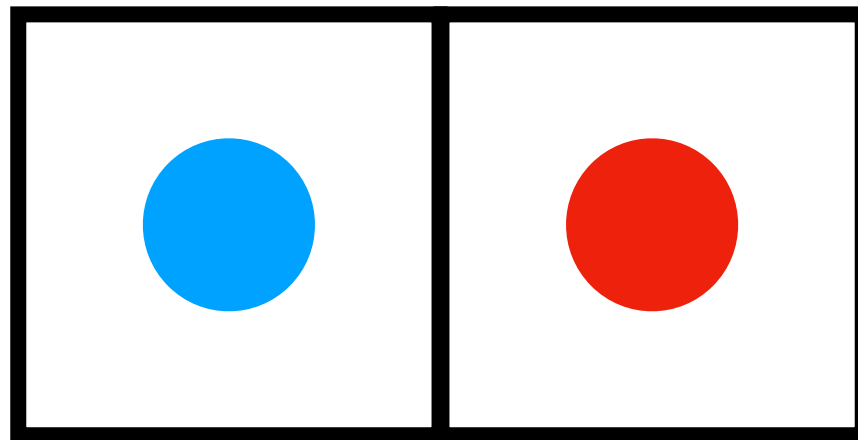


(cons 1 2)

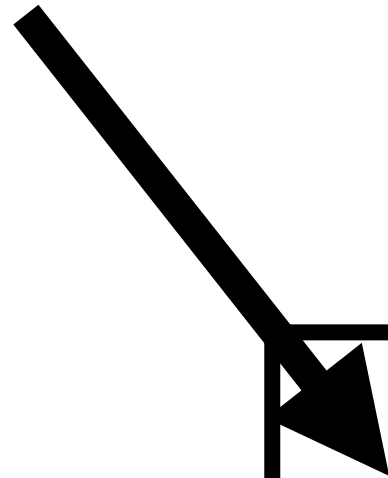


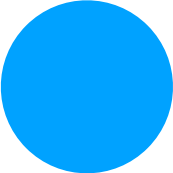

cons is the the natural
constructor of the language

I use two strange words to refer to the
elements of this cons cell



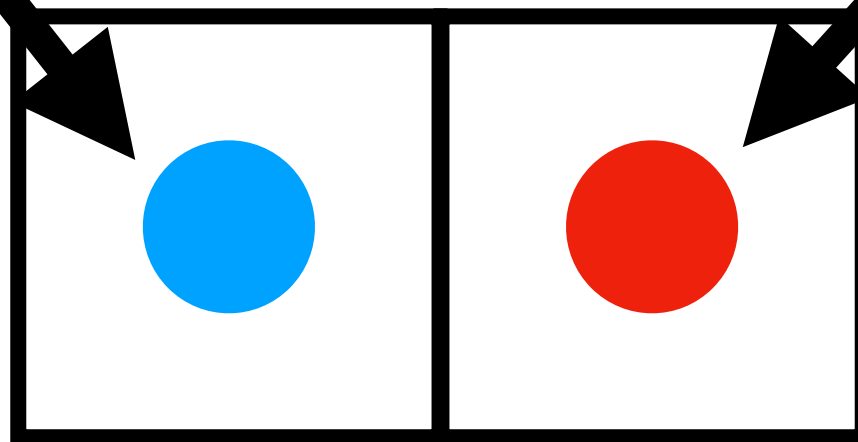
“car”



	
---	---

“car”

“cdr”



Because car and cdr break apart what I build with
cons, I call them my **destructors**

And that's all

And that's all

Atoms 'sym 23 #\c

Empty list '()

cons (cons 'sym 23)

car/cdr (car (cons 'sym 23))

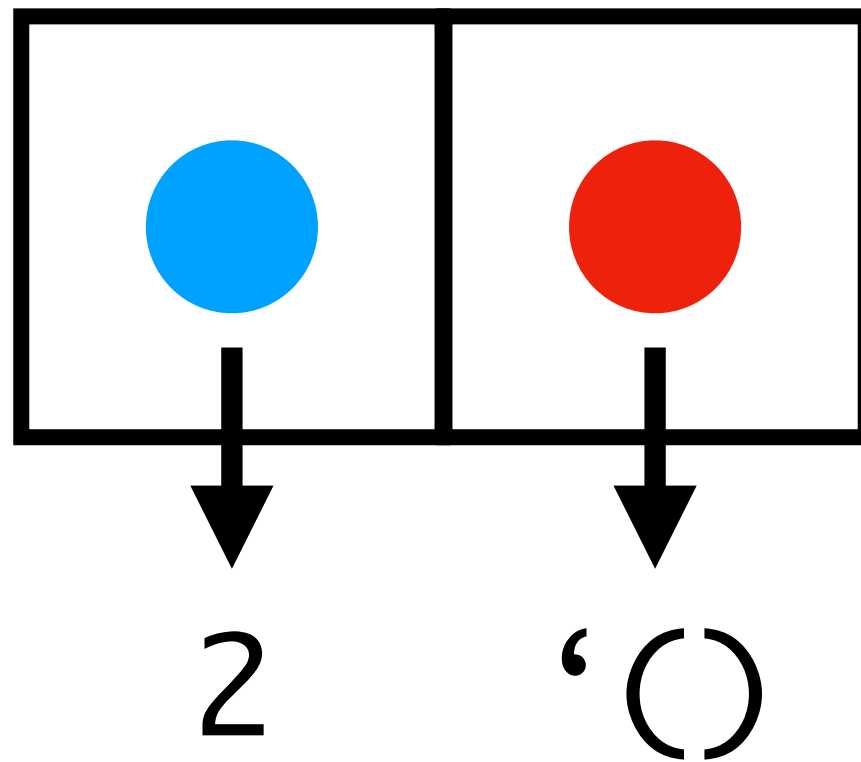
Using just this, I can make a list

Using just this, I can make a list

(And everything else in the world, but we'll
get back to that...)

If I want to make the list containing 2 I do this

(cons 2 '())



When I do this, Racket prints it out as a list

‘ (2)

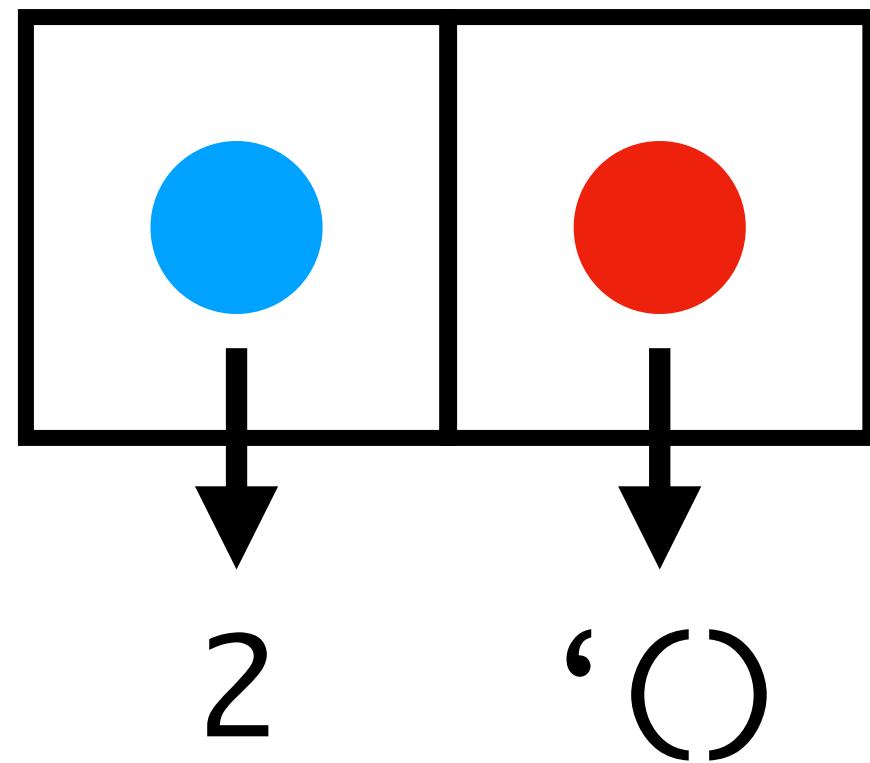
The way to read this is

“The list containing 2, followed by the empty list.”

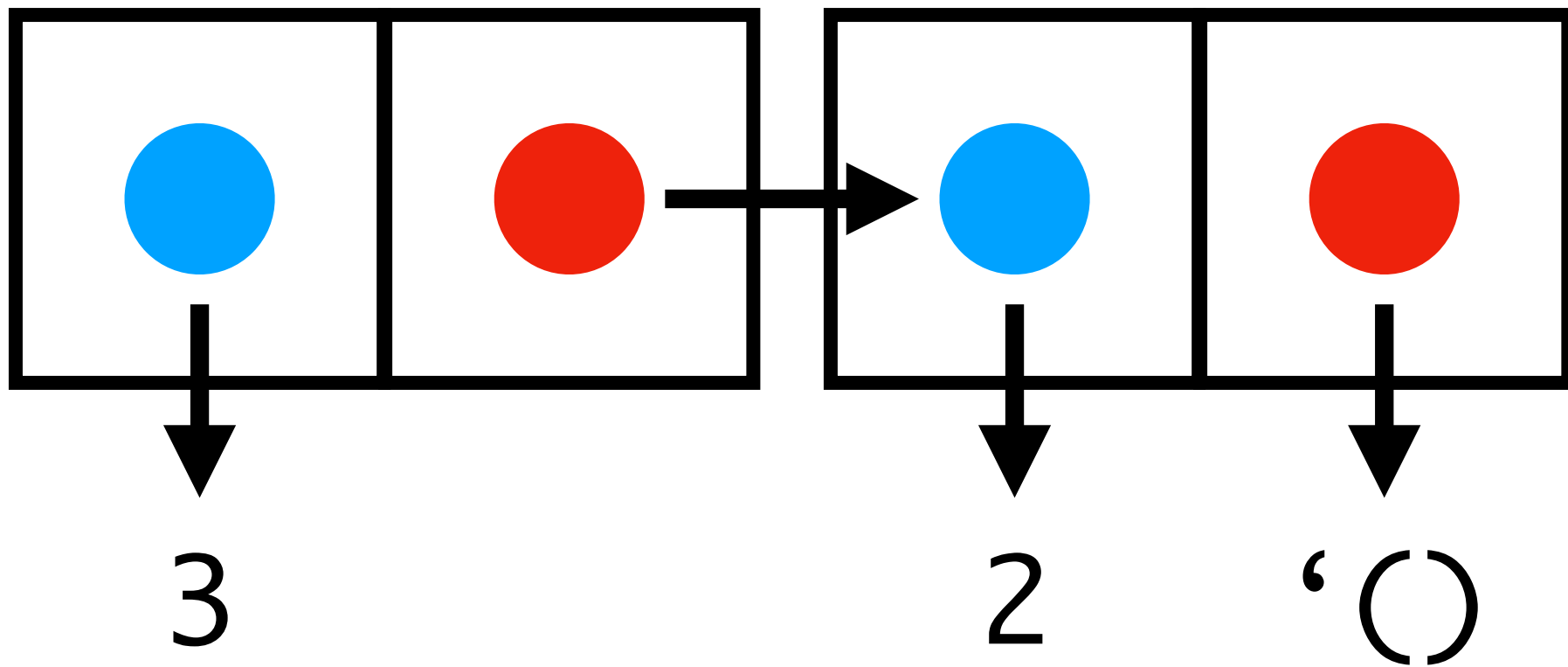
Just as I can build lists of a single element, I can build
larger lists from smaller lists...

And I do that by stuffing lists inside other lists...

(cons 2 '())



(cons 3 (cons 2 '()))



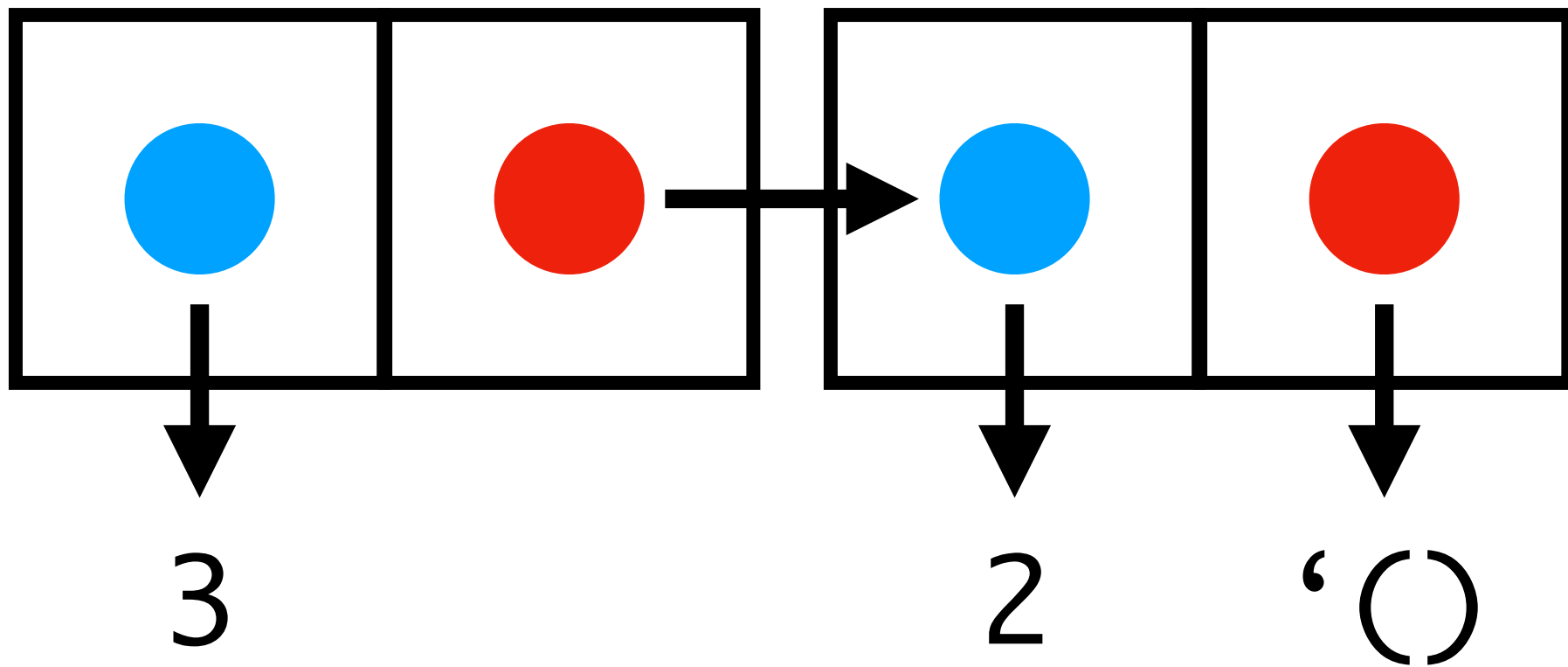
Racket will print this out as

‘(3 2)’

Of course, I probably need at least numbers
as primitives right?

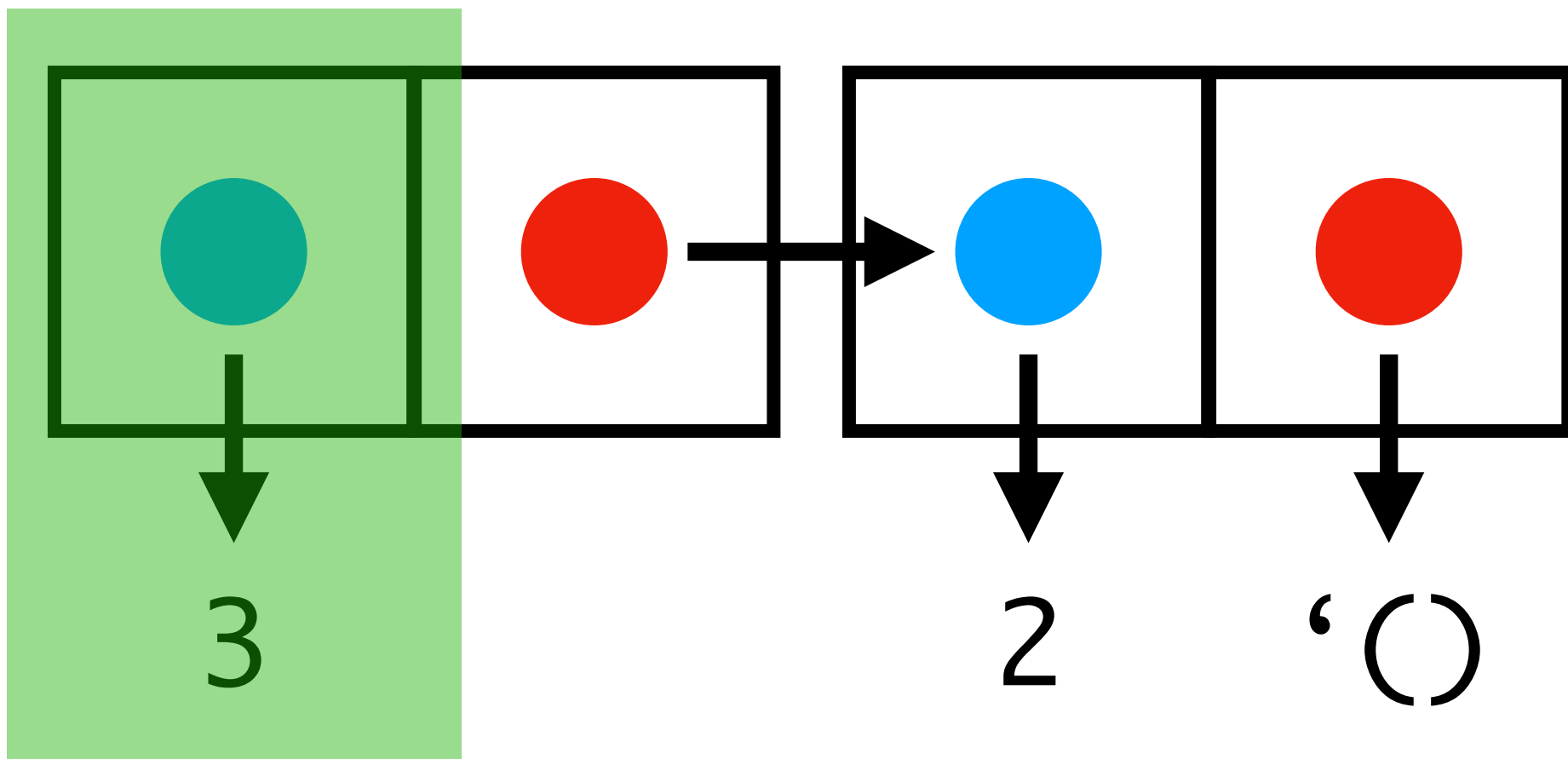
To get the head of a list, I use `car`

(cons 3 (cons 2 '()))



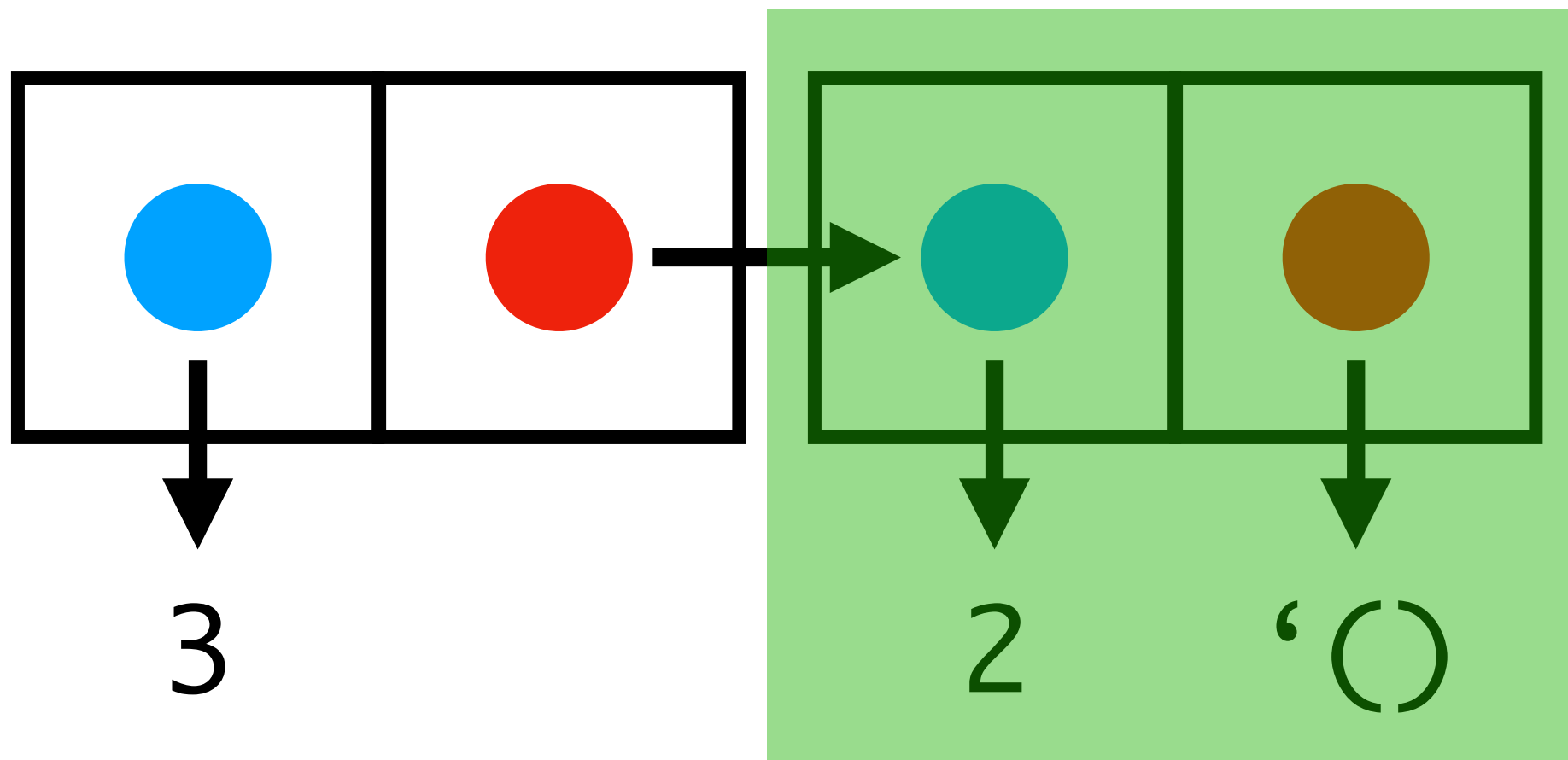
(car

(cons 3 (cons 2 '()))))



(cdr

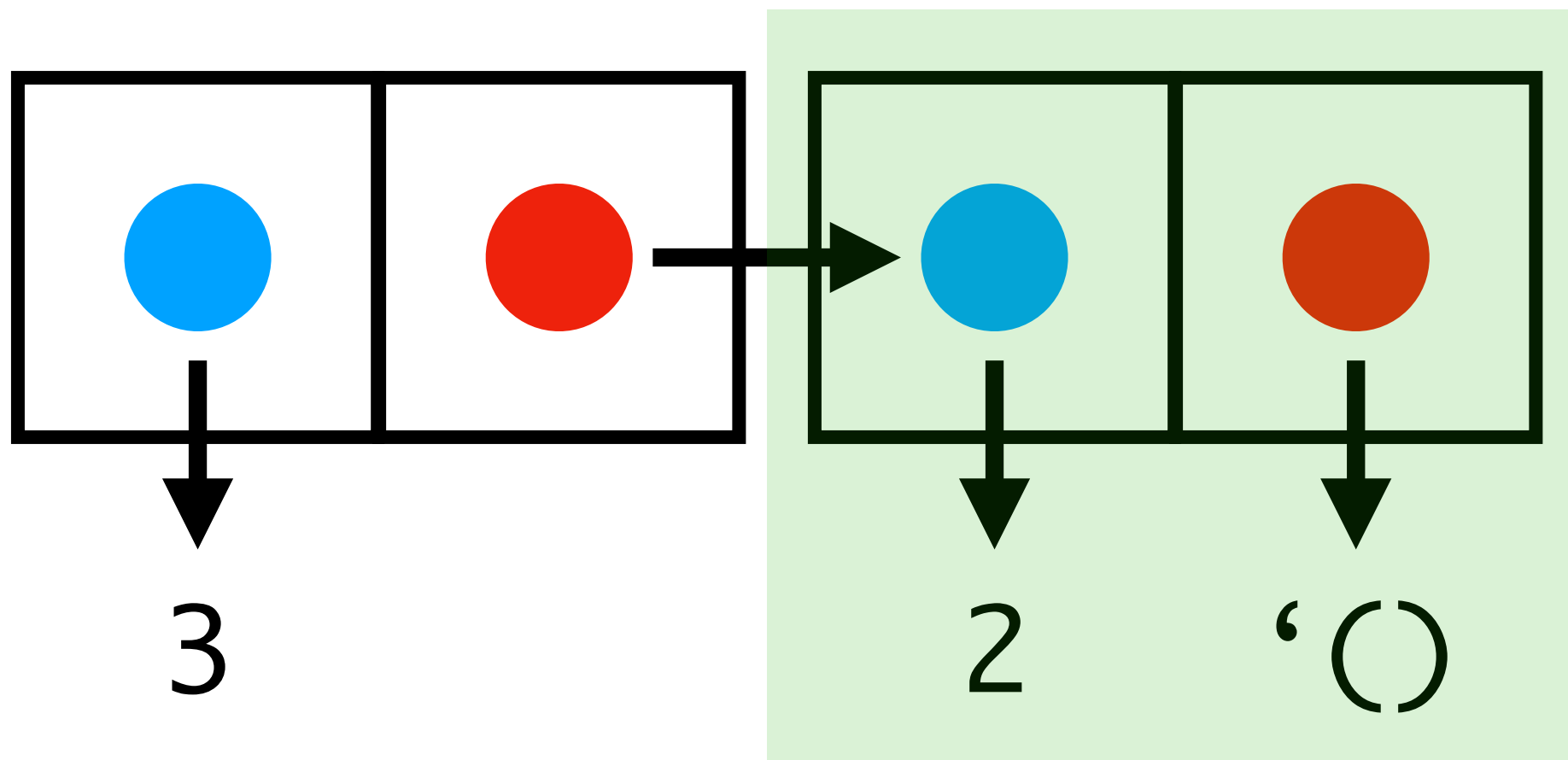
(cons 3 (cons 2 '()))))



So now how would I get the second element?

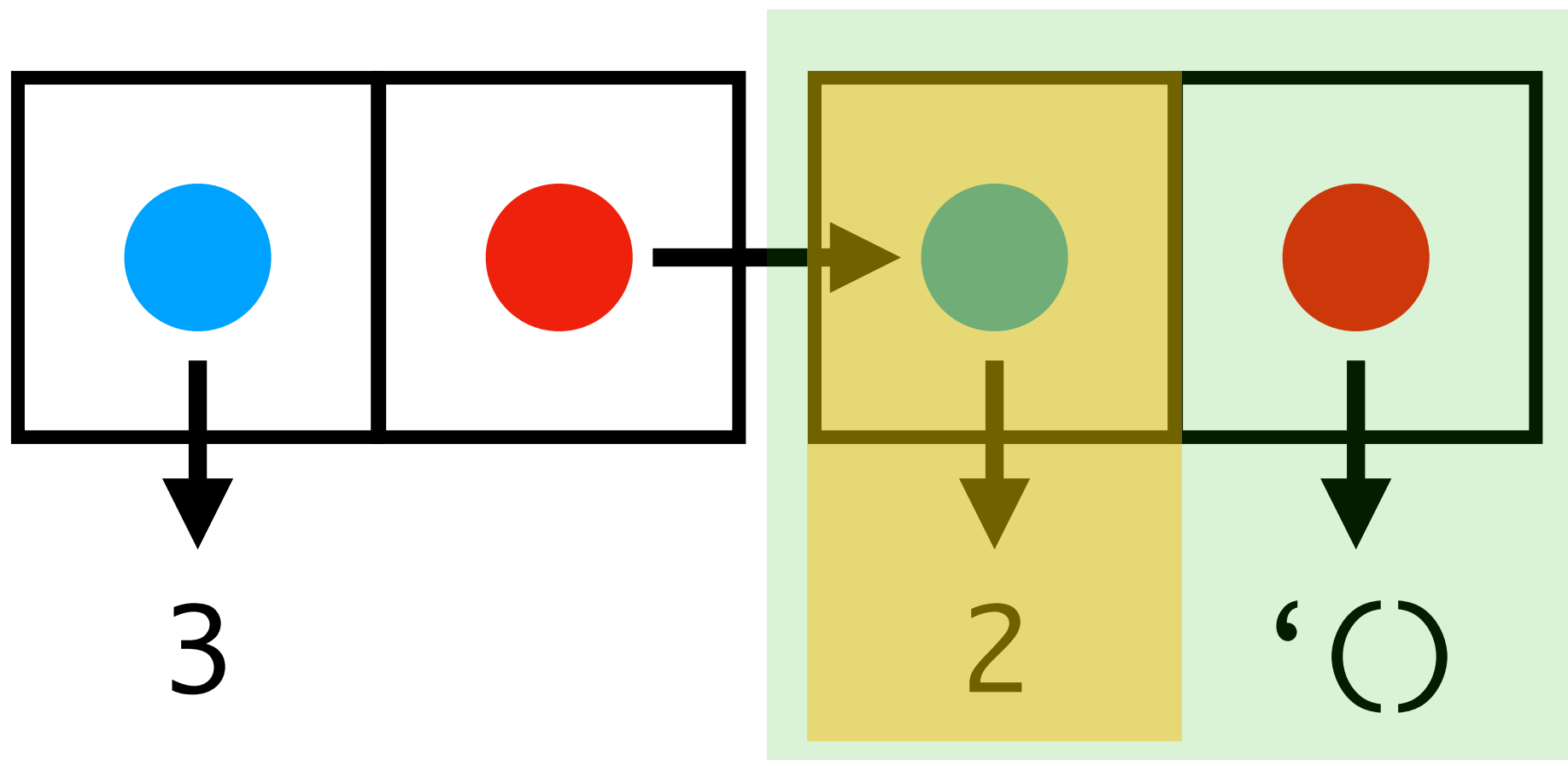
(cdr

(cons 3 (cons 2 '()))))



(car
(cdr

(cons 3 (cons 2 '()))))



Racket abbreviates

`(cons 1 (cons 2 (cons... (cons n '())...)))`

as...

`'(1 2 ... n)`

If I wanted to write out lists, I could do so using

```
(cons 1 (cons 2 ...))
```

How do I get the nth element of a list?

```
(define (nth list n)
  (if (= 0 n)
      (car list)
      (nth (cdr list) (- n 1))))
```

Now, write `(map f l)`

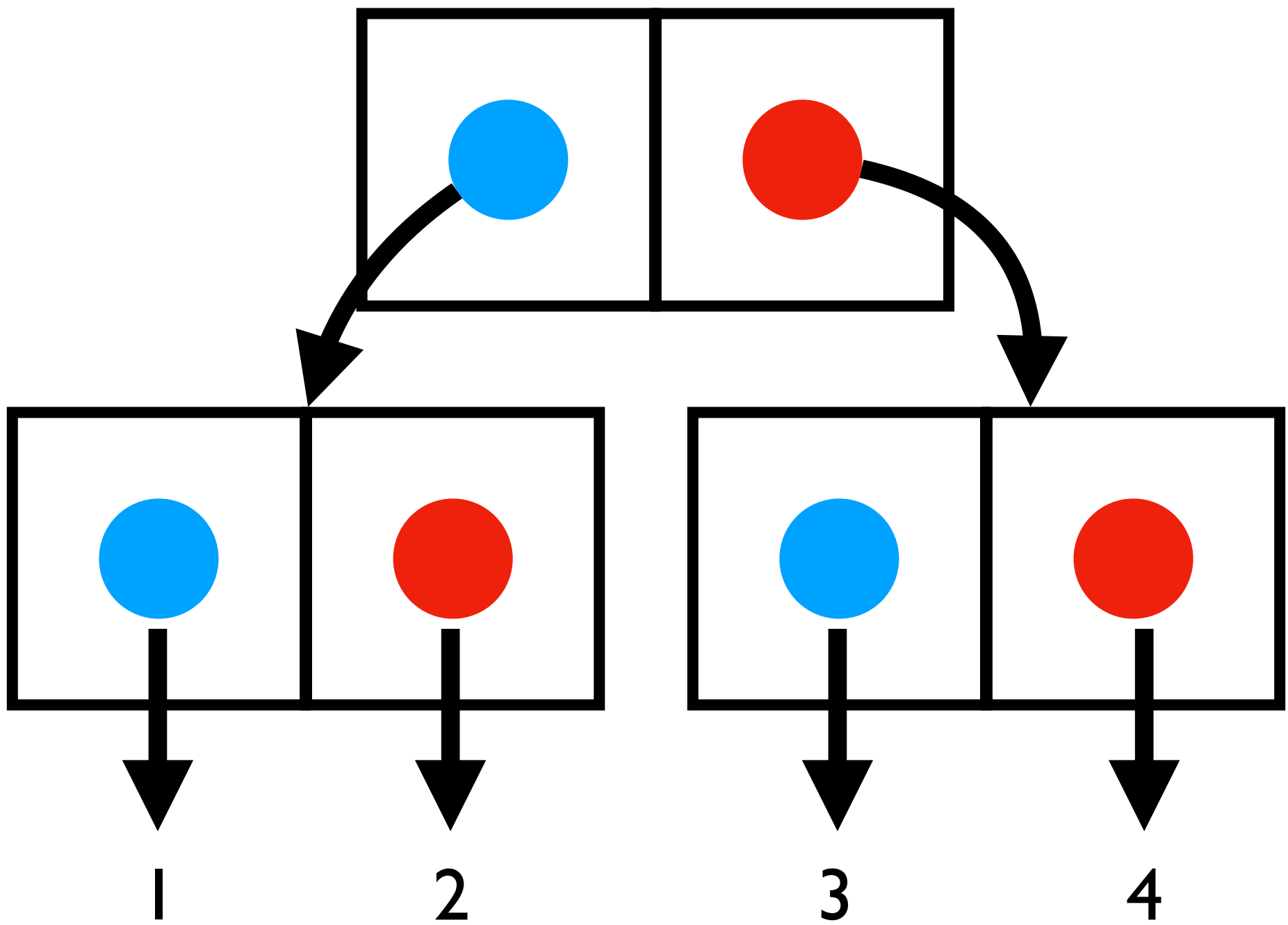
Writing lists would get quite laborious

Instead, I can use the primitive function **list**

```
(list 1 2 'serpico)
```

```
'(1 2 serpico)
```

Oh, and actually I can use this to represent trees too



How would I build this?

```
(define empty-tree 'empty-tree)
```

```
(define (make-leaf num) num)
```

```
(define (make-tree left right)  
  (cons left right))
```

You define (left-subtree tree)

```
(define (least-element tree)
  (if (number? tree)
      tree
      (least-element (left-subtree tree))))
```


But surely I need things like numbers right?

It turns out, you could build those using just
cons, car, cdr, if, =, and '()

Define the number n as ...

'()

'(())

'(())

...

```
(define (weird-plus i j)
  (if (equal? i '())
      j
      (weird-plus (cdr i)
                   (cons '() j)))))
```

(weird-plus '(() ()) '(() ()))
 '(() () () ())

It turns out, if I'm clever, we can even get rid of
if and **equal**

(Though we shall not do so here..)

I can build my own datatypes in this manner

I usually write **constructor** functions to help
me build datatypes

I usually write **constructor** functions to help
me build datatypes

And I usually write **destructor** functions to
access it

```
(define (make-complex real imag)  
  (cons real imag))
```

And I usually write **destructor** functions to
access it

```
(define (make-complex real imag)
  (cons real imag))
```

```
(define (get-real complex)
  (car complex))
```

```
(define (get-imag complex)
  (cdr complex))
```

Now, define `(add-complex c1 c2)`

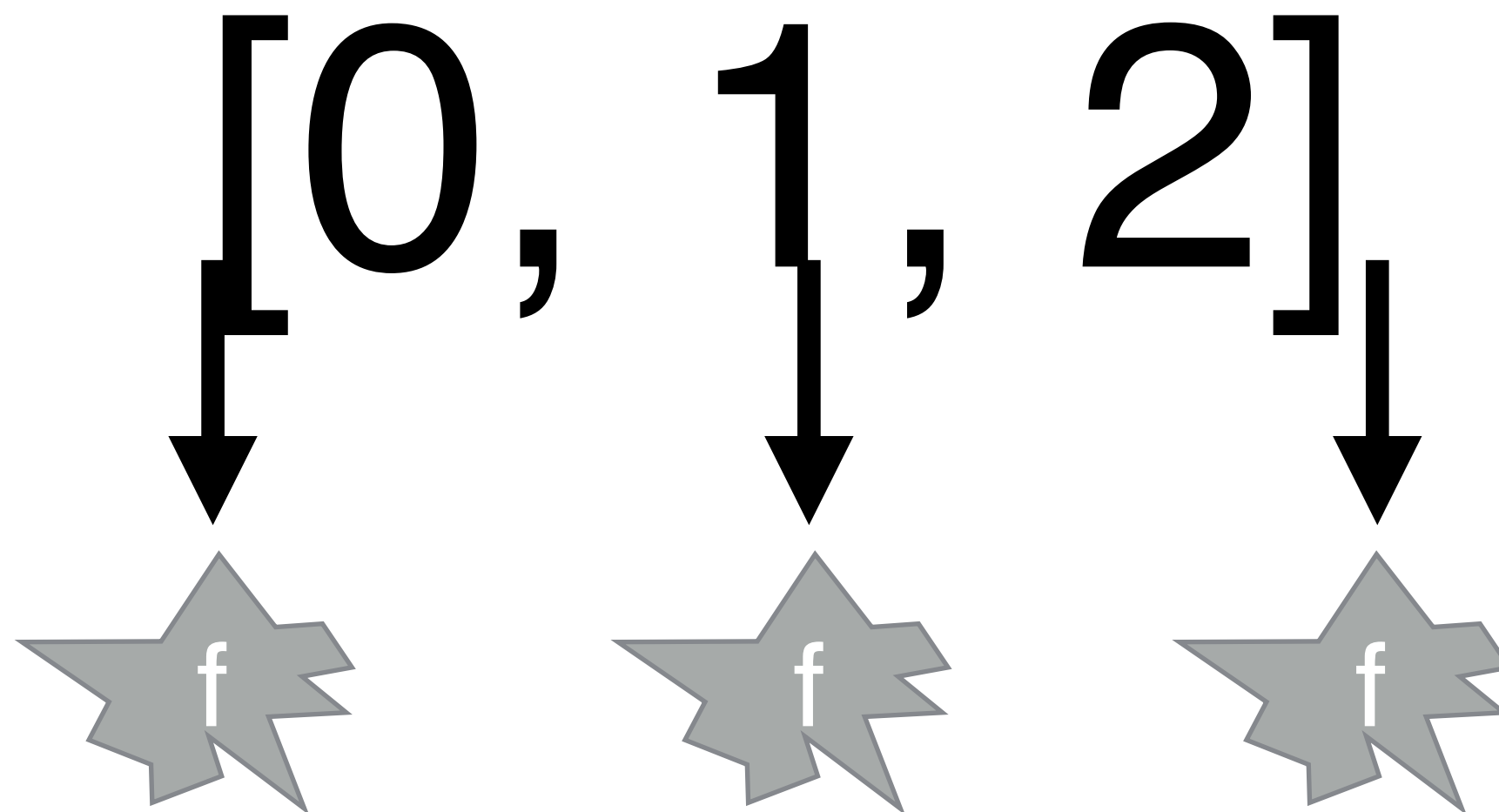
Next, define `(make-cartesian x y)`

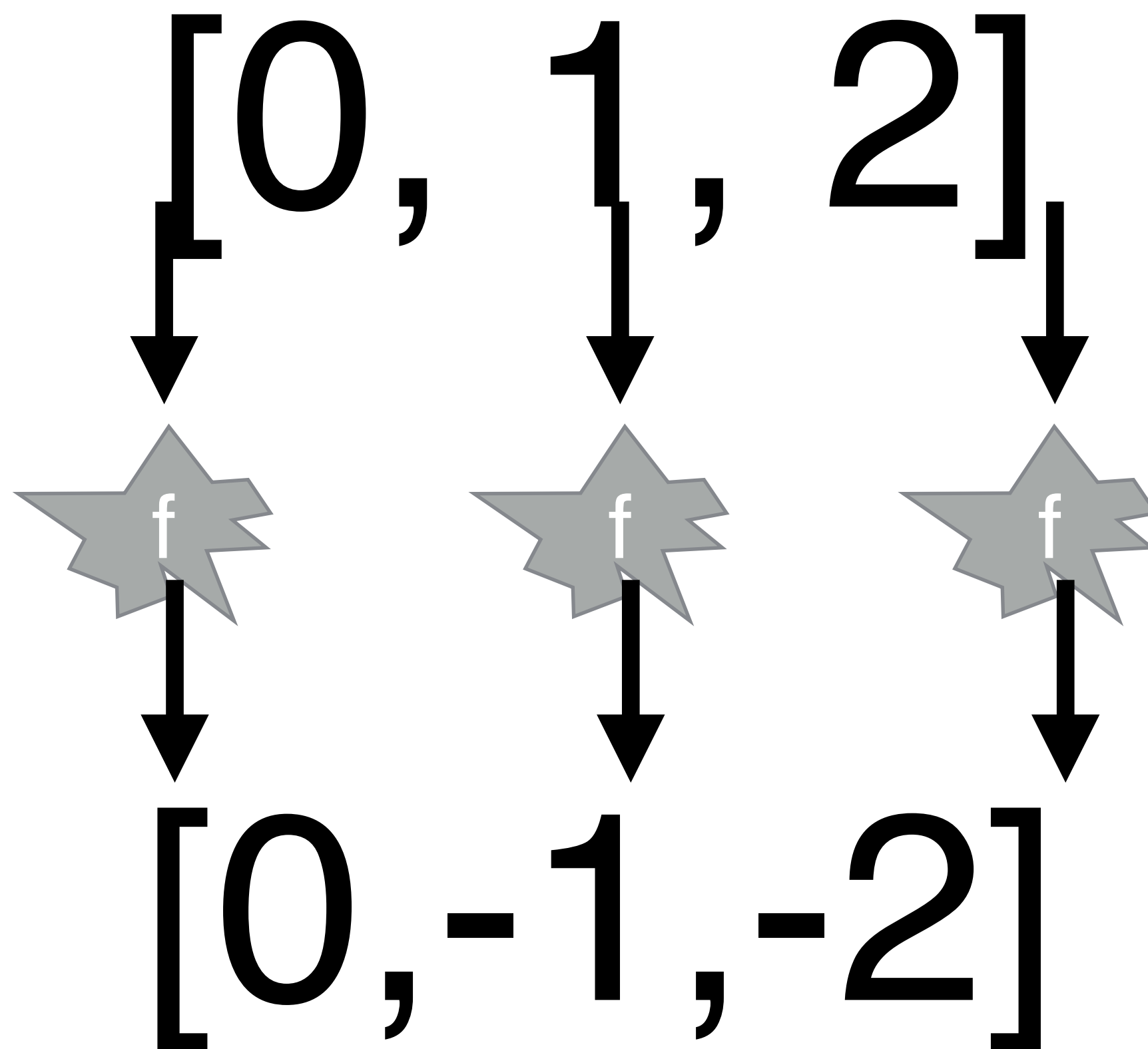
And the associated helper functions

```
> (map (lambda (str) (string-ref str 0)) '("ha" "ha"))  
'(#\h #\h)
```

(map f l) takes a function f and
applies f to each element of l

[0, 1, 2]





Next class we will talk about...

struct

match

I/O

Intermediate Racket Programming

Tail Recursion

Tail recursion is the way you make recursion fast
in functional languages

Anytime I'm going to recurse more than 10k
times, I use tail recursion

(I also do it because it's a fun mental exercise)

Tail Recursion

A function is tail recursive if all recursive calls are in tail position

A call is in tail position if it is the last thing to happen in a function

The following is not tail recursive

```
(define (factorial x)
  (if (equal? x 0)
      1
      (* (factorial (- x 1)) x)))
```

The following is tail recursive

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
```

The following is not tail recursive

```
(define (factorial x)
  (if (equal? x 0)
      1
      (* (factorial (- x 1)) x)))
```

Explain to the person next to you why this is

The following is tail recursive

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
```

Swap. Explain to the person next to you why this is

This isn't merely trivia!

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

factorial 2 1

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

factorial 2 1

>factorial 1 2

factorial 1 2

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

factorial 2 1

>factorial 1 2

factorial 1 2

>factorial 0 2

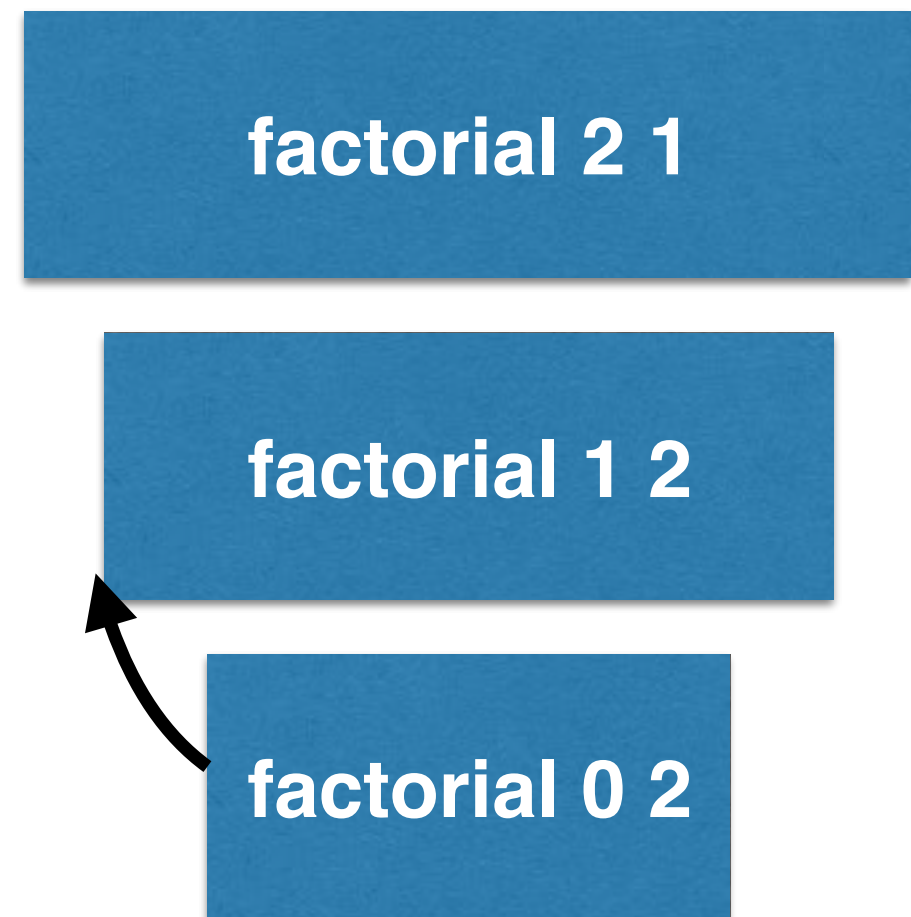
factorial 0 2

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

>factorial 1 2

>factorial 0 2

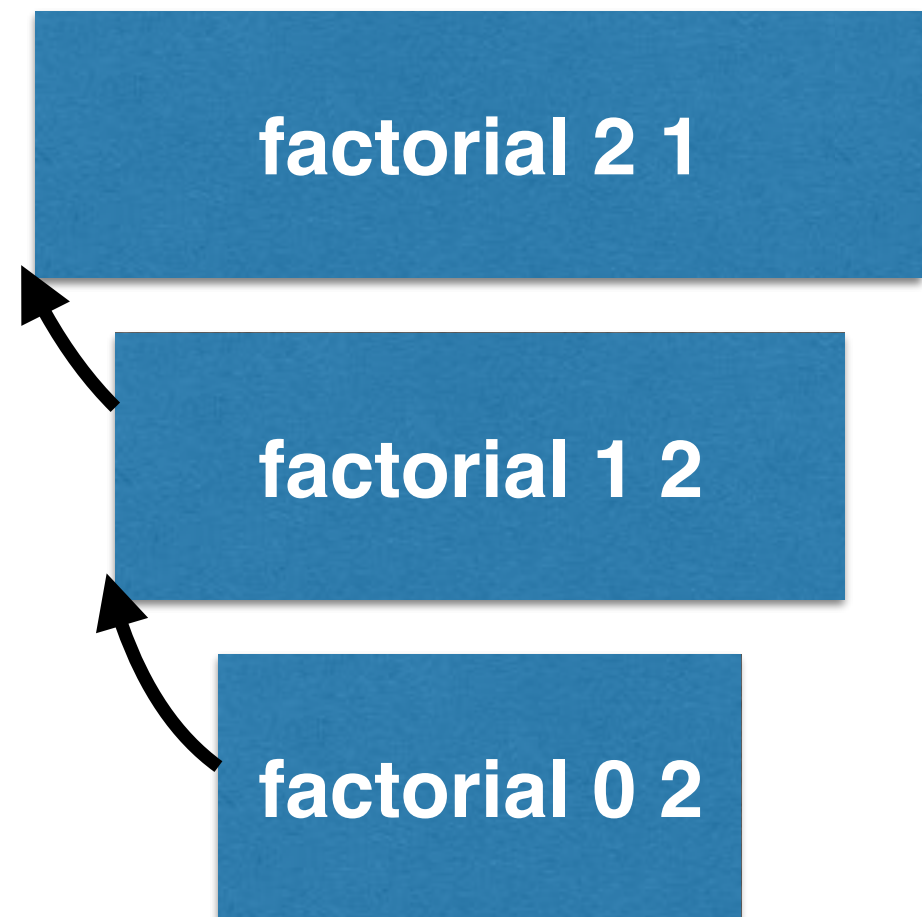


```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
; .. Later
(factorial 2 1)
```

>factorial 2 1

>factorial 1 2

>factorial 0 2



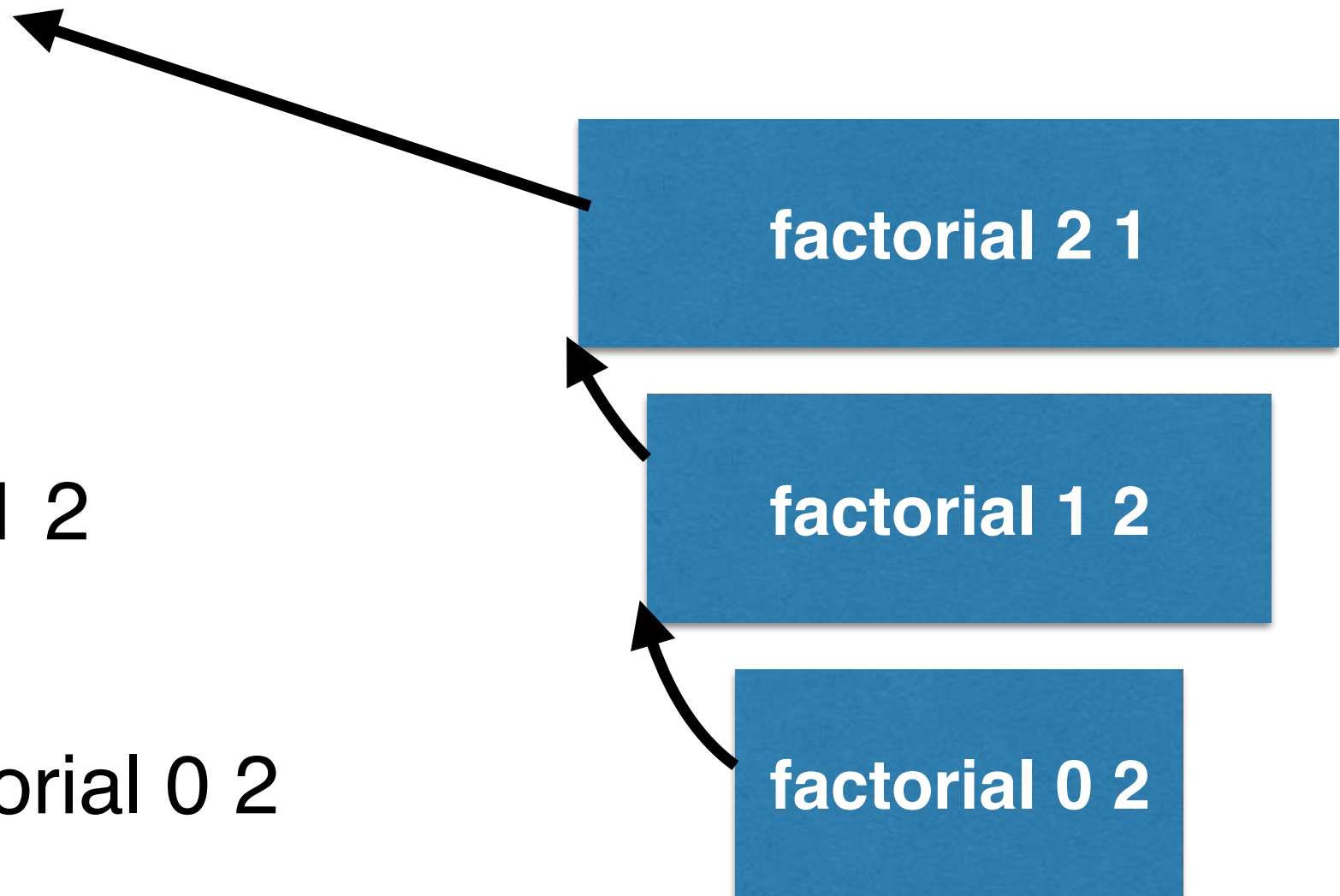

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
```

```
; .. Later
(factorial 2 1)
```

```
>factorial 2 1
```

```
>factorial 1 2
```

```
>factorial 0 2
```



```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
```

```
; .. Later
(factorial 2 1)
```

But wait!
I don't need the stack at all!

```
>factorial 2 1
```

```
>factorial 1 2
```

```
>factorial 0 2
```



factorial 2 1



factorial 1 2



factorial 0 2

Insight: in tail recursion, the stack is just used for copying back the results

So just forget the stack. Just give the final result to the original caller.

Insight: in tail recursion, the stack is just used for copying back the results

So just forget the stack. Just give the final result to the original caller.

Insight: in tail recursion, the stack is just used for copying back the results

This is called “tail call optimization”

```
(define (factorial x acc)
  (if (equal? x 0)
      acc
      (factorial (- x 1) (* acc x))))
```

; .. Later

```
(factorial 2 1)
```

>factorial 2 1

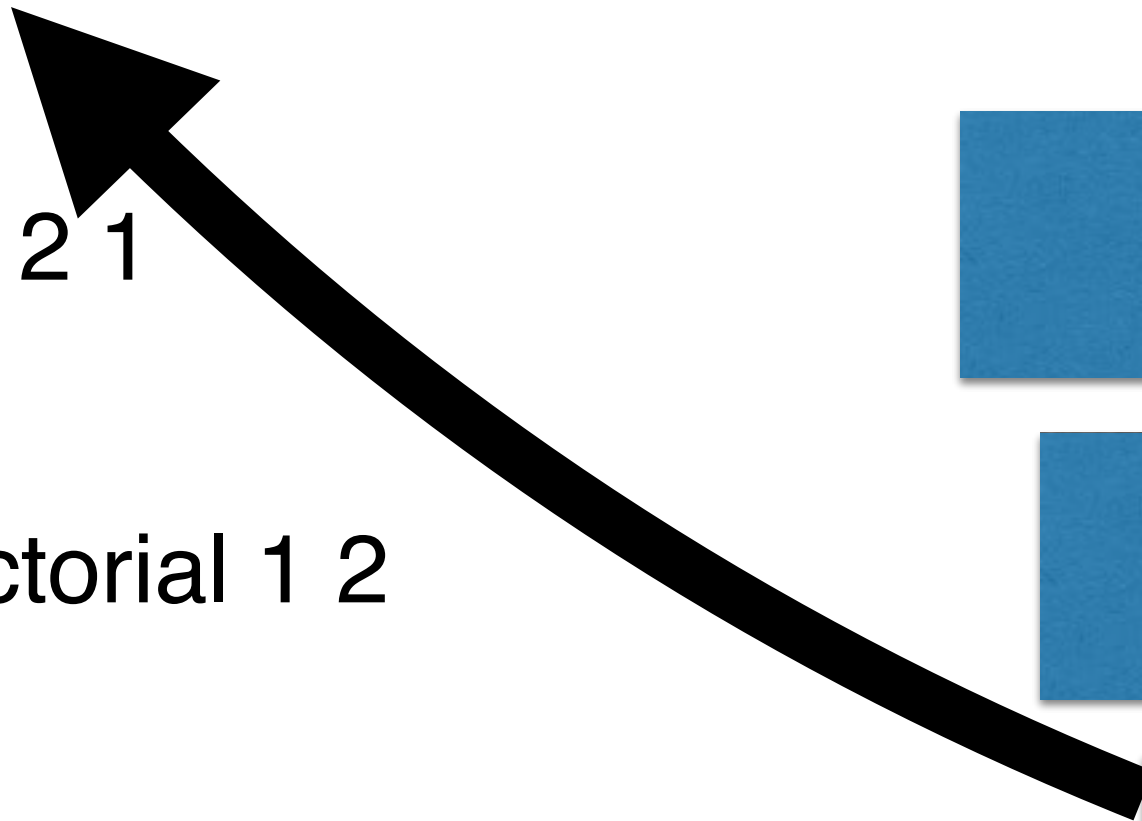
>factorial 1 2

>factorial 0 2

factorial 2 1

factorial 1 2

factorial 0 2



Why couldn't I do that with this?

```
(define (factorial x)
  (if (equal? x 0)
      1
      (* (factorial (- x 1)) x)))
```

Talk it out with neighbor

Tail recursion for λ and profit...

To make a function tail recursive...

- add an extra accumulator argument
- that tracks the result you're building up
- then return the result
- might have to use more than one extra arg
- Call function with base case as initial accumulator

This isn't the only way to do it, just a nice trick
that usually results in clean code...


```
(define (factorial x)
```

```
  (if (equal? x 0)
```

```
    1
```

```
    (* (factorial (- x 1)) x)))
```



```
(define (factorial-tail x acc)
```

```
  (if (equal? x 0)
```

```
    acc
```

```
    (factorial-tail (- x 1) (* acc x))))
```

```
(define (factorial x) (factorial-tail x 1))
```

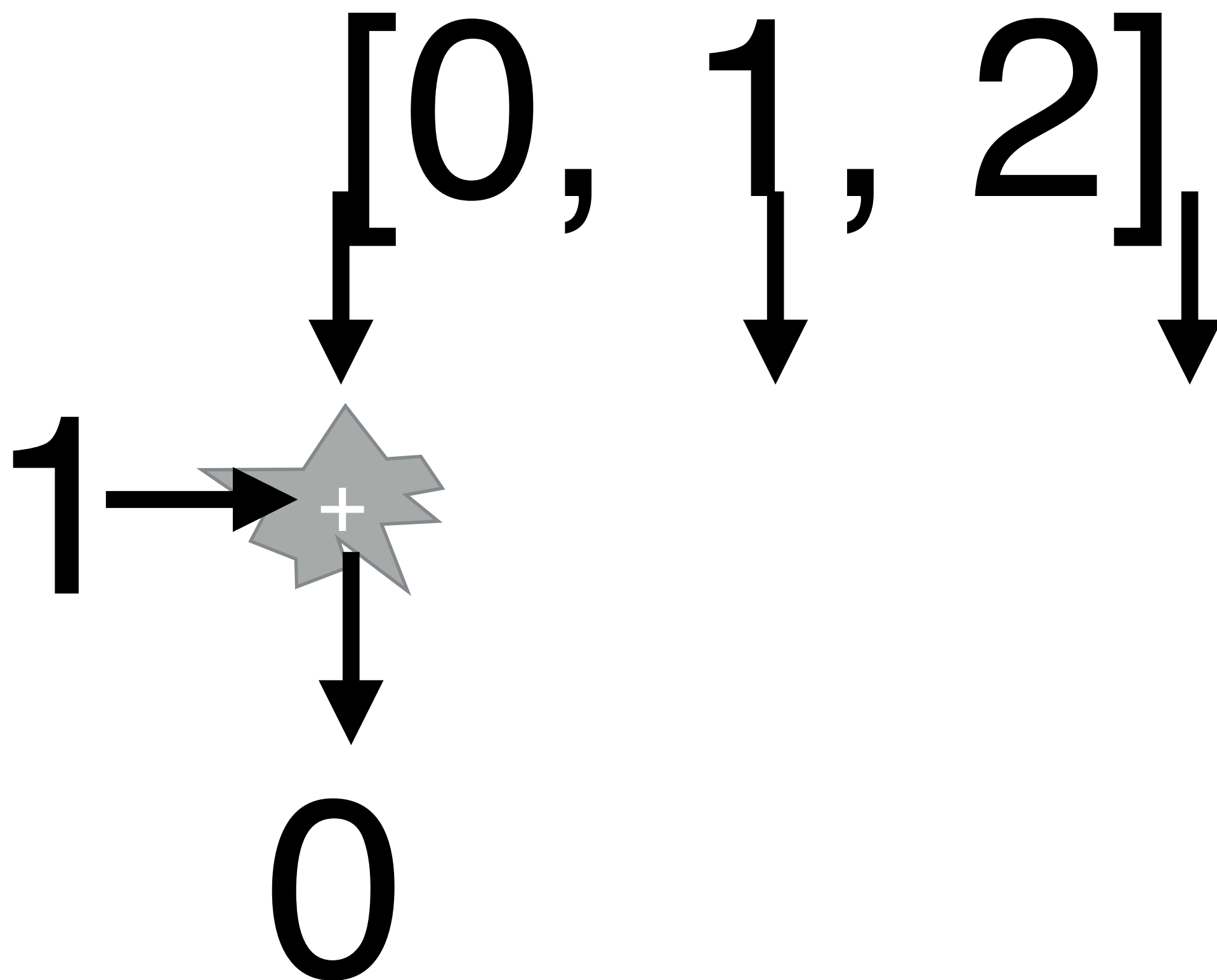
```
(define (max-of-list l)
  (cond [(eq? (length l) 1) 1]
        [(empty? l) (raise 'empty-list)]
        [else (max (first l) (max-of-list (rest l)))]
  )))
```

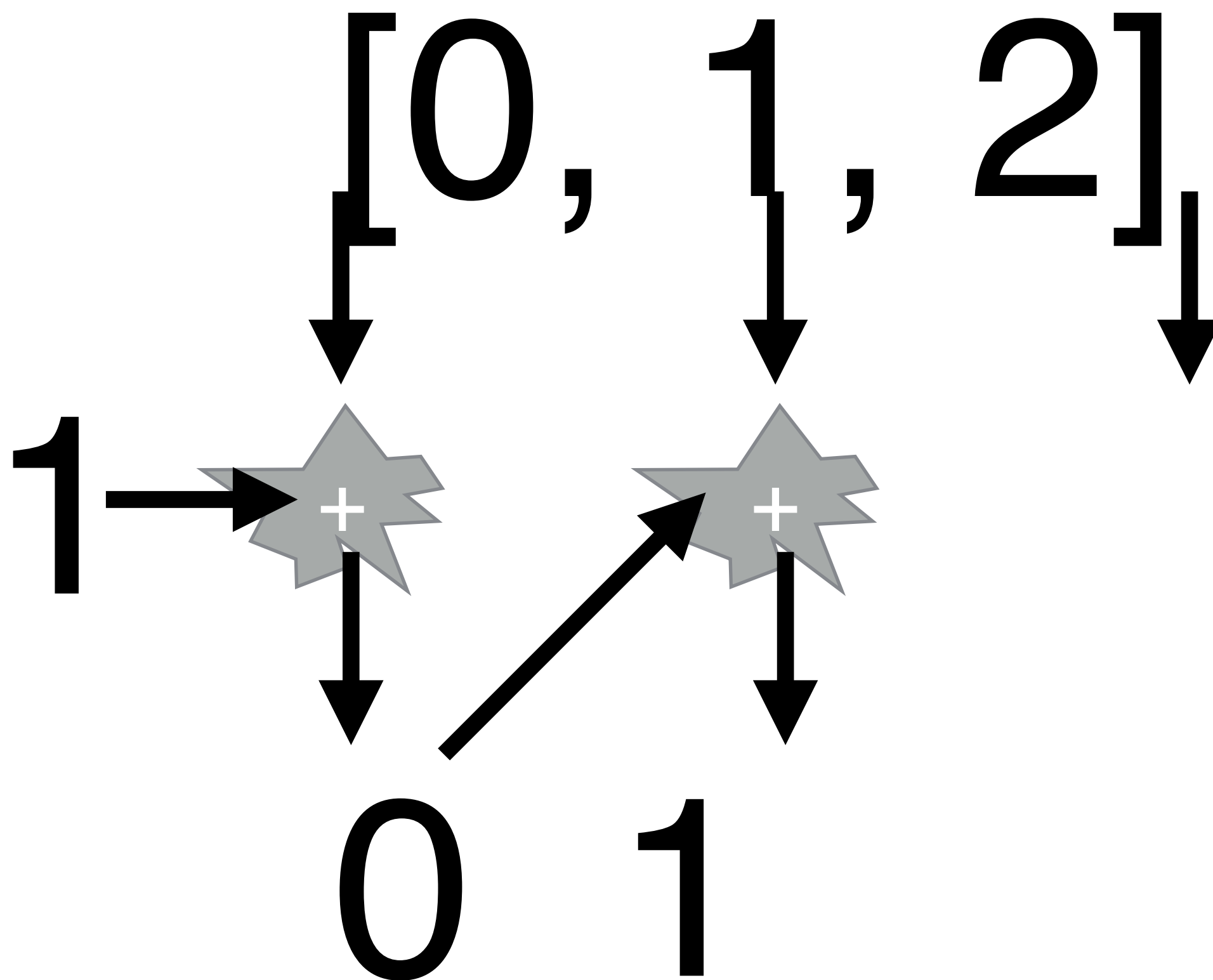
Write this as a tail-recursive function

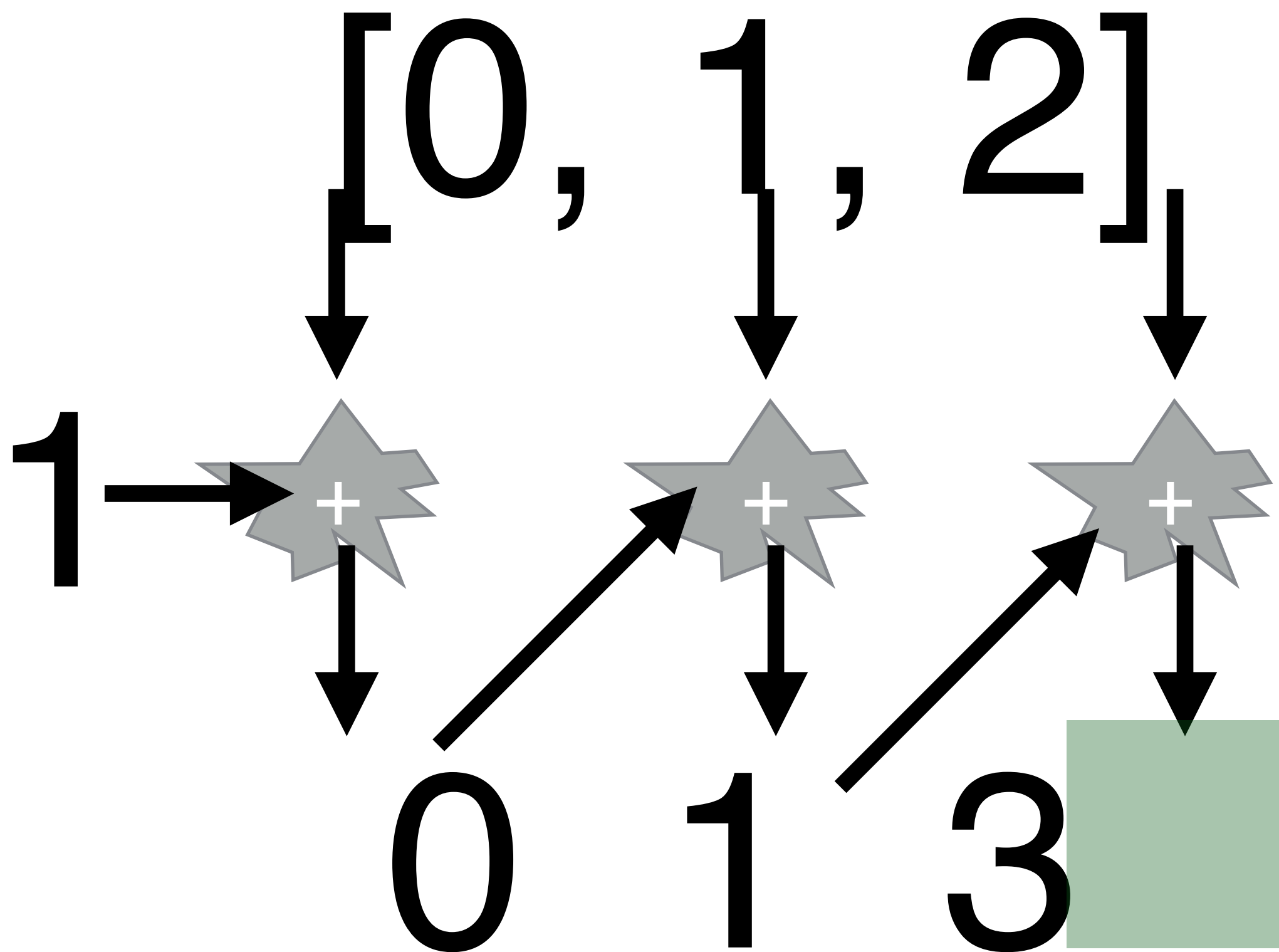
foldl

Like map, a higher order function operating on lists

$$(\text{foldl } / \ 1 \ ' (1 \ 2 \ 3)) = (/ \ 3 \ (/ \ 2 \ (/ \ 1 \ 1)))$$
$$(\text{foldl } + \ 0 \ ' (1 \ 2 \ 3)) = (+ \ 3 \ (+ \ 2 \ (+ \ 1 \ 0)))$$







```
(define (concat-strings l)
  (foldl (lambda (next_element accumulator)
    (string-append next_element accumulator))
    ""
    (reverse l)))
```

Challenge: use foldl to define max-of-list

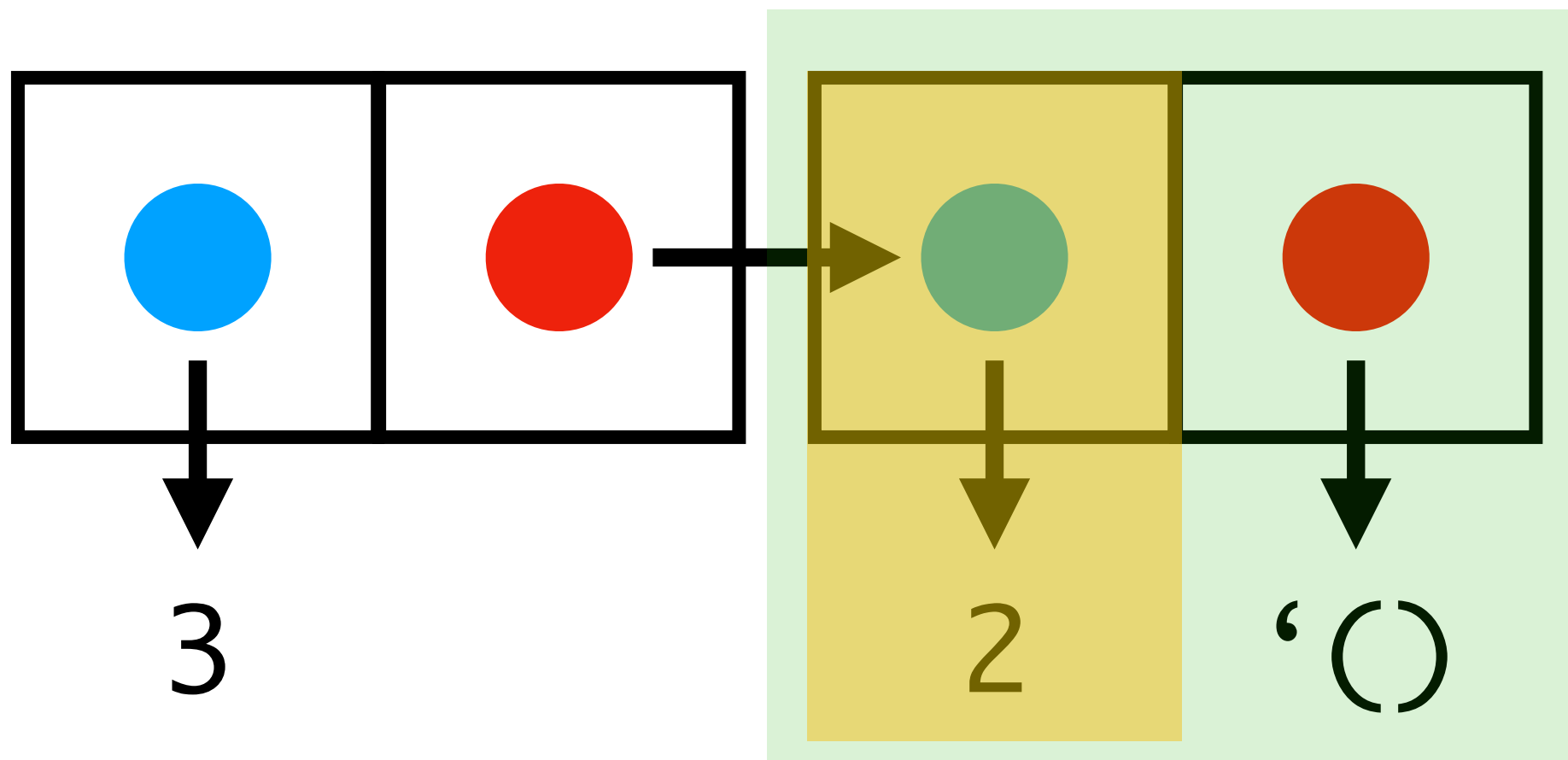
****Challenge: define foldl**

Structures, Pattern Matching, and Contracts

Last time

(car
(cdr

(cons 3 (cons 2 '()))))



This time

5 Programmer-Defined Datatypes

New datatypes are normally created with the `struct` form, which is the topic of this chapter. The class-based object system, which we defer to [Classes and Objects](#), offers an alternate mechanism for creating new datatypes, but even classes and objects are implemented in terms of structure types.

5.1 Simple Structure Types: `struct`

To a first approximation, the syntax of `struct` is

```
(struct struct-id (field-id ...))
```

Examples:

```
| (struct posn (x y))
```

The `struct` form binds *struct-id* and a number of identifiers that are built from *struct-id* and the *field-ids*:

- *struct-id* : a *constructor* function that takes as many arguments as the number of *field-ids*, and returns an instance of the structure type.

Example:

```
| > (posn 1 2)  
| #<posn>
```

- *struct-id?* : a *predicate* function that takes a single argument and returns `#t` if it is an instance of the structure type, `#f` otherwise.

Examples:

Use **struct** to define a new datatype

(struct empty-tree ())

(struct leaf (elem))

(struct tree (left right))

Copy these

```
(struct empty-tree ())
```

```
(struct leaf (elem))
```

```
(struct tree (value left right))
```

(empty-tree)

(leaf 23)

(tree 12 (empty-tree) (leaf 23))

Racket automatically generates helpers...

tree?

tree-left

tree-right

Write max-of-tree

Use the helpers

Pattern matching

Pattern matching allows me to tell Racket the
“shape” of what I’m looking for

Manually pulling apart data
structures is laborious

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```


Variables are bound in the match, refer
to in body

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Note: match struct w/ (name params...)

```
(define (max-of-tree t)
  (match t
    [(leaf e) e]
    [(tree v _ (empty-tree)) v]
    [(tree _ _ r) (max-of-tree r)]))
```

Define is-sorted

Can match a list of x's

(list x y z ...)

(1 2 3 4)

x = 1 y = 2 z = '(3 4)

Can match cons cells too...

(cons x y)

Variants include things like match-let

10

Racket has a “reader”

(read)

Racket “reads” the input one *datum* at a time

```
> (read)
```

```
(1 2 3)
```

```
'(1 2 3)
```

```
> (read)
```

```
1 2 3
```

```
1
```

```
> (read)
```

```
2
```

```
> (read)
```

```
3
```

```
>
```

Read will “buffer” its input

NETFLIX

7%



Loading

(read-line)

(open-input-file)

Contracts


```
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Write out the call and return type of this
for yourself

```
(define (factorial i)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

What are the call / return types?

What is the pre / post condition?

```
(define (gt0? x) (> x 0))
```

```
(define/contract (factorial i)
  (-> gt0? gt0?)
  (cond
    [(= i 1) 1]
    [else (* (factorial (- i 1)) i)]))
```

Now in tail form...


```
(define (fac-tail i)
  (letrec ([h (lambda (i acc)
                (cond
                 [(= i 0) acc]
                 [else (h (- i 1) (* acc i))])]))
    (h i 1)))
```

Now, let's say I want to say it's equal to
factorial...

```
(define/contract (fac-tail i)
  (->i ([x (>=/c 0)])
    [result (x) (lambda (result) (= (factorial x) result))])
  (letrec ([h (lambda (i acc)
    (cond
      [(= i 0) acc]
      [else (h (- i 1) (* acc i))]))])
    (h i 1)))
```

```
(->i ([x (>=/c 0)])  
      [result (x) (lambda (result) (= (factorial x) result))])
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

```
(define/contract (reverse-string s)
  (-> string? string?)
  (list->string (reverse (string->list s))))
```

$$(\leq/c \ 2)$$

`<=` / C takes an argument `x`, returns a function `f` that takes an argument `y`, and `f(y) = #t` if `x <= y`

\leq / C takes an argument x , returns a function f that takes an argument y , and $f(y) = \#t$ if $x \leq y$

(Note: \leq / c is also doing some bookkeeping, but we won't worry about that now.)

Challenge: write `<=/c`

Three stories




```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (string-append (f s1) (f s2)))
```

```
(define (reverse-string s)
  (list->string (reverse (string->list s))))
```

Scenario: you call call-and-concat with reverse

Scenario: you call call-and-concat with
reverse, 12, and “12”

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```


Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

What went wrong?

Now define

```
(define/contract (call-and-concat f s1 s2)
  (-> (-> string? string?) string? string? string?)
  (length (string-append (f s1) (f s2))))
```

What went wrong?

Who is to blame?